Group Mutual Exclusion to Scale Distributed Stream Processing Pipelines

Mehdi Belkhiria
Univ Rennes. Inria, CNRS, IRISA
Université Rennes 1
Rennes, France
mehdi.belkhiria@irisa.fr

Marin Bertier
Univ Rennes. Inria, CNRS, IRISA
INSA Rennes
Rennes, France
marin.bertier@irisa.fr

Cédric Tedeschi
Univ Rennes. Inria, CNRS, IRISA
Université Rennes 1
Rennes, France
cedric.tedeschi@irisa.fr

***Abstract*—Stream Processing has become the *de facto* standard way of supporting real-time data analytics. Stream Processing applications are typically shaped as pipelines of operators, each record of the stream traversing all the operators of the graph. The placement of these operators on nodes of the platform can evolve through time according to different parameters such as the velocity of the input stream and the capacity of nodes. Such an adaptation calls for mechanisms such as dynamic operator scaling and migration. With the advent of Fog Computing, gathering multiple computationally-limited geographically-distributed resources, these mechanisms need to be decentralized, as a central coordinator orchestrating these actions is not a scalable solution any more.**

In a fully decentralized vision, each node hosts part of the pipeline. Each node is responsible for the scaling of the operators it runs. More precisely speaking, nodes trigger new instances of the operators they runs or shut some of them down. The number of replicas of each operator evolving independently, there is a need to maintain the connections between nodes hosting neighbouring operators in the pipeline. One issue is that, if all these operators can scale in or out dynamically, maintaining a consistent view of their neighbours becomes difficult, calling for synchronization mechanisms to ensure it, to avoid routing inconsistencies and data loss.

In this paper, we show that this synchronization problem translate into a particular Group Mutual Exclusion (GME) problem where a group comprises all instances of a given operator of the pipeline and where conflicting groups are those hosting neighbouring operators in the pipeline. The specificity of our problem is that groups are fixed and that each group is in conflict with only one other groups at a time. Based on these constraints, we formulate a new GME algorithm whose message complexity is reduced when compared to algorithms of the literature, while being able to ensure a high level of concurrent *occupancy* (the number of processes of the same group in the critical section (the scaling mechanism) at the same time.

***Keywords*-stream processing; scaling; group mutual exclusion;**

I. Introduction

With the recent growing need to timely process large volumes of data, Stream Processing (SP) becomes a dominant programming paradigm to extract knowledge out of continuous streams of data. Stream Processing Engines (SPEs) constitute the practical cornerstone of stream processing systems. SPEs are typically toolboxes offering high-level programming models and APIs easing the task of the programmer willing to develop and deploy stream processing applications at scale. Within SPEs, an application is commonly expressed as a set of operators that each record of the stream traverses. These operators can be represented by directed acyclic graphs (DAGs), representing the order in which items traverse the operators. Very often, these DAGs are simple chains (*a.k.a. pipelines*).

To deal with the varying velocity of the input stream, SPEs include scaling facilities: Operators are scaled in and out: if the velocity increases at an operator's entry, the number of compute *nodes* (typically processes or vCPUs) supporting it should increases accordingly. Conversely, if the velocity drops, the number of computing units is reduced. Figure 1 illustrates scaling in an SP application: the initial application is composed of 5 operators in a pipeline. At some point (b), the middle operator is scaled out so as to match the increased stream velocity (for the sake of simplicity, we assume other operators do not have to be scaled out.) The scaling process results in the operator being deployed over more nodes. However, because the set of nodes for this operator changed, the nodes hosting neighbouring operators in the pipeline (predecessor and successor operators) has to update their routing tables and establish the connections with the newly introduced nodes. This is mandatory to ensure that the input stream of the scaled out operator is balanced amongst the nodes supporting it. Later (c), the velocity of the input stream of the same operator decreases and one node is removed. To avoid sending data to a removed node which may lead to data loss, the routing tables of the neighbouring nodes need to be updated accordingly.

Scaling in a stream processing context has been studied extensively in a centralized context, where a single manager process is responsible for dynamically adjust the computing power dedicated to each operator [12], [17], [18], [31]. Recently, few works proposed solutions to decentralize this scaling process, so as to improve its scalability over geographically-dispersed computing platforms such as the Fog [7], [10], [11], [6], [29]. In these solutions, the operators take decisions to scale (in or out) independently, leading to potential concurrency issues: consider two neighbouring groups of instances taking independent yet concomitant scaling decisions, as depicted in Figure 2: a third instance

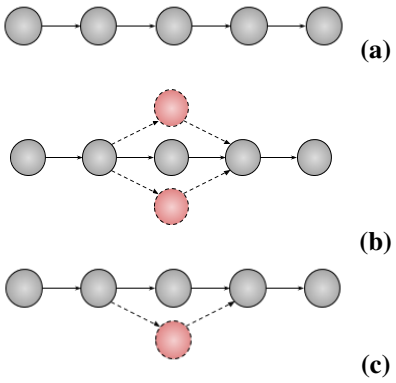

Figure 1. (a) A simple pipeline. (b) The third operator is scaled out on two extra nodes. Its predecessors and successors in the pipeline update their routing table with the new nodes. (c) The velocity of the input stream of the third operator decreases. Some of the nodes hosting it are stopped. The routing tables of neighbours are updated accordingly.

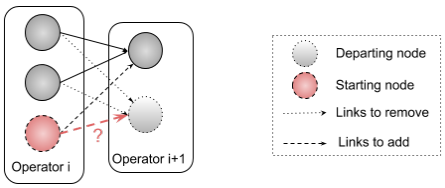

Figure 2. Concurrent neighbouring scaling processes.

is spawned for Operator i while one of the two instances of Operator $i+1$ is stopped. These works still present a certain degree of centralization: instances of a given operators are coordinated through local *schedulers* or *leaders*. Also, these works avoid the concurrency issues by temporarily shutting down the system during reconfiguration, which, in our case, want to avoid: scaling should not interrupt data processing.

In the fully decentralized vision we adopt in this paper, each node needs to maintain its set of successors, at the same time, the same set of successors is actually evolving. If not handled properly, this may lead to incorrect views. For instance, the new node may *believe* that the node being removed is still alive. This calls for synchronisation mechanisms: two neighbouring groups of instances cannot scale concurrently without facing inconsistencies in their routing tables which in turn can lead to abnormal communications and data loss.

While our previous work proposed an ad-hoc synchronisation protocol including acknowledgements [5], a more generic solution is needed. The scaling of neighbouring operators in the pipeline cannot take place at the same time to ensure thread safety in the scaling process. In other words, when an operator scales, the scaling of neighbouring operators have to be postponed. As we place this work in a fully-decentralized context, all instances of a given operator may start duplicating or terminating itself at any time. This translates into a Group Mutual Exclusion (GME) problem [21]: The critical resource is here the scaling operation, and it can be requested by any node at any time. Nodes running the same operator are considered as one group. Within one group, all nodes can scale at the same time without any risk and are even encouraged to do so, but two groups hosting neighbouring operators need to enter the scaling process / critical section in a sequential manner.

The problem also shows some similarity with the local mutual exclusion problem: the pipeline encodes the conflicts: only neighbouring nodes (or groups) are conflicting. In this sense, put aside the groups, this problem is similar to the dining philosopher's problem [16], or the more generic local resource allocation problem [8]. Because we assume there is no cycle in the pipeline (thus reducing the risk of deadlocks), the problem appears to be simpler. Altogether, our problem translates into a local GME problem, where a group is in conflict with only two other groups: its predecessor and successor groups in the pipeline. A simple distributed test can ensure that a group wishing to enter the critical section (the scaling process) can solve the conflict with its two neighbours sequentially. For instance, a group can first secure the access to the scaling with its predecessor group, and once it is secured can start requesting the access to the scaling with its successor group. If done sequentially over the two groups in conflict, the core problem to be solved is a GME problems where only two groups are in conflict.

One difficulty however lies in the fact that scaling in and out dynamically adds and removes processes in groups, new processes being able in their turn to start another local scaling round. In this paper, we assume a set of stable nodes within each group which are responsible for the scaling. In other words, the process of ensuring mutual exclusion between groups of neighbours is fully decentralized but groups are fixed. In other words, while new nodes can appear dynamically, we assume that the process of coordination between groups is delegated to a set of core nodes responsible for it. These nodes cannot disappear and form the *core* group.

In this paper, we propose an algorithm for the GME problem, which has been built having the specific constraints of its applications to decentralized scaling in stream processing applications. Assuming that only two fixed groups are in conflict, our algorithm benefit from the fact that processes do not need to communicate with every other process but only with those which are in the other group. In other words, when compared with other GME algorithms, our algorithm generates less messages than other algorithms made for nodes where nodes can move from one group to another one, while exhibiting a similar concurrency level (the number of nodes from the same group concurrently in critical section).

Section II positions the present work by reviewing the literature about mutual exclusion with a focus on Group Mutual Exclusion and Local Mutual Exclusion. Section III presents the algorithm and gives a proof of its safety and

liveness. Section IV shows a validation of the algorithm, in particular when compared with other algorithms from the literature. Section V concludes.

II. Related Work

Mutual exclusion in distributed settings are mostly solved by two families of algorithms: permission-based and token-based. In permission-based algorithms, such as the Ricart-Agrawala algorithm [27], processes typically send a request to a set of other processes when they want to enter the critical section, and enter it only when they have received a positive acknowledgement from all processes in this set. To ensure liveness (lockout freedom), processes maintain a sequence number according to Lamport's causality rule [23] which allows a global ordering of processes in case of a conflict. When looked at as a resource allocation problem, mutual exclusion can be modeled as a graph where processes are vertices and edges represent conflicts over resources. Chandy and Misra proposes to make this graph acyclic to ensure one process can be distinguished in case of multiple concomitant demands, thus removing the need for timestamps [13]. A special case of this approach is the well-known *dining philosophers* problem where the graph is a ring, and where different simple strategies can be used to ensure liveness [15]. The set of process to ask the permission to can be reduced by the notion of *quorums*. With quorums, in contrast with the initial Ricart-Agrawala algorithm, receiving the permission of only a subset of all processes is enough to enter critical section, provided these quorums and their intersections are well defined [25], [1].

In token-based protocols, the safety of mutual exclusion is ensured by having a *token* travelling amongst the processes: the right to enter critical section is materialized by the possession of the token [24]. Two main strategies have been proposed for the token's movement. Either the token is perpetually moving and thus will reach any process in a finite time, or it is asked. Such algorithms were proposed first on a ring [26], and then generalized to any topology [19].

Group Mutual Exclusion is also referred to as the *congenial talking philosophers* (CTP) problem [20], [22], [21]. Group Mutual Exclusion generalizes basic mutual exclusion (if we consider groups of 1 process) and other classical concurrency problems such as readers/writers [14]. Joung proposes two permission-based algorithms to solve the problem [21]. In this problem, philophers share a room of limited capacity. Each philosopher alternates between thinking and talking in a forum taking place in the room. Each philosopher can choose dynamically what forum to attend, but only one forum can take place at a time. A philosopher can successfully enter a forum when the room is empty or when another philosopher attending the same forum is already in the room. The first proposed by Joung (RA1) is a direct adaptation of the Ricart-Agrawala algorithm. This first algorithm suffers from a poor concurrent occupancy of the room: As soon as a philosopher with a higher priority wants to attend a forum which is not the currently forum in the room, no more philosopher will be able to attend this forum before it is closed. To counteract this problem inherited from Ricart-Agrawala, Joung proposes Algorithm RA2 in which once a philosopher enters a forum, it becomes a captain for this forum and can *capture* processes that could not attend it in RA1 because of philosophers with a higher priority. RA2 is our closest related work: the present work also bypasses the total ordering of processes by allowing processes to enter the critical section in spite of their lower priority. Yet this bypassing is allowed by the conflicting group and not by members of our own group. More generally, in contrast with RA2, our algorithm, provided there are only two fixed groups, does not need processes to communicate with members of the same group.

The problem has also been tackled in the specific case of a ring network [30]. Similarly, the authors present several improvements over an adaptation of the Ricart-Agrawala algorithm for basic mutual exclusion for GME in rings. Other works dealt with GME over tree networks [4]. Inspired by the works about standard mutual exclusion mentioned before, the GME problem has been solved using tokens (in the particular case of the ring) [9], and using quorums [32], [2].

III. A Mutual Exclusion Protocol for Two Fixed Groups

A. System Model

Nodes (instances of operators) communicate by *reliable* and *FIFO* asynchronous message passing (a message reaches its destination in a finite time, and two messages sent through the same channel are processed in the same order they were sent). The nodes cannot crash or leave during the algorithm execution. Each node belongs to a fixed group (the operator it instantiates), and the group composition can not change.

B. The 2-FGME Algorithm

Our algorithm, named 2-FGME for *Two Fixed Groups Mutual Exclusion* is given by the pseudo-code in Algorithms 1 and 2. 2-FGME, as Joung's RA2 algorithm, relies on the basic principle promoted by the Ricart / Agrawala algorithm for standard mutual exclusion and extended for GME in Joung's RA1 algorithm: when a node wants to enter the critical section, it sends a request message to all of its *competitors*. Yet in our case, to reduce the traffic, competitors are limited to the set of nodes in the other group. Nodes maintain a Lamport's clock included in the requests to represent the priority of a request and globally ordering the queries.

The second important aspect is that, consequently, the notion of captains cannot be used, as it requires nodes within the same group to communicate together. Thus we rely on another idea to ensure concurrent occupancy: When Node n_i

receives a request from Node n_j, it acknowledges it either because it comes from a higher priority node, or because it already authorized a node within its competitors, even if n_j has a lower priority than n_i. A node stops acknowledging requests from its competitors when one of them either leaves the critical section or postpones its own request. Doing so, the algorithm prevents the two groups from authorizing each others or having one continuously acknowledging request from the other group.

1) Initialisation and locally maintained sets.: Initially, as represented by the INITIALISATION procedure (see Line 1), a node is in the IDLE state, its Lamport clock LC is 0, and the other group is not authorized (represented by the *competitors_authorized* boolean variable). Notice the three sets of nodes maintained by each node:

- *waiting_reqs* contains the nodes for which a request has been received but was not acknowledged yet.
- *reqs_to_send* contains the nodes to which a request has to be sent: when a node has its other group authorized, it postpones the sending of its requests until the other group is not authorized anymore
- *acks_to_ignore* contains the nodes for which an acknowledgements is to be ignored. This comes from the fact that, when a node *a posteriori* authorizes nodes in the other groups, the potential acknowledgements coming from those nodes are not valid anymore.

2) Messages.: The algorithm relies on three message types.

- The *Req(n_i, LC_i)* message is the message sent to the competitors when a node wants to enter the critical section. It contains two parameters: i) n_i, the unique id of the sending node, and ii) its Lamport clock LC_i.
- The *Ack(n_i, renew, end)* message represents an acknowledgement from Node n_i. It has two extra boolean parameters. If $renew$ is set, it means that in spite of n_i acknowledging a demand, it is a reminder that n_i is still requesting the critical section too. $renew$ is typically set when a requesting node authorizes *a posteriori* a node with a lower priority but in the other (authorized) group. If end is set, it expresses the fact that the node acknowledges the demand because leaving the critical section.
- The *End(n_i, LC_i)* message expresses a status similar to the $Ack(end)$ message but is intended to be sent to nodes for which no acknowledgement is needed (typically because no requests are pending for these nodes).

3) Requesting the critical section: When Node n_i requests the critical section, it applies the algorithm in Lines 8-17. There are two cases:

1) The other group is not authorized, in which case, *Req* is sent to the n_i's competitors.
2) The other group is authorized (n_i typically acknowledged at least one demand prior to its own demand). In this case, the request is postponed (until the other is not authorized anymore): n_i reminds that it needs to send its own request by adding its competitors in the *reqs_to_send* set.

4) Receiving a request: Upon receiving a demand (a *Req* message) from $n_{i'}$ on n_i, there are two cases when to acknowledge the request:

1) n'_i has a higher priority than n_i. In this case, n_i applies the pseudo-code in Lines 20-40. First, due to the higher priority of the demand, n_i sends a simple Ack message (without setting the *renew* or *end* flags). The rest of this case is to take into account that the other group is now authorized. After setting the *competitors_authorized* variable, n_i *a posteriori* authorizes nodes amongst competitors that previously sent their queries but where initially not authorized (typically because of their lower priority compared to that of n_i's demand). This is materialized in Line 28 by sending an *Ack* to all the nodes in *waiting_reqs*. Also, an *End* message is sent to the competitors for which no requests were received to withdraw n_i demand (on Line 40). Recall that this is a temporary n_i withdrawing: n_i still wants to enter the critical section. n_i holds its demand until the other group is not authorized anymore: n_i add the competitors in $reqs\_to\_send$. Finally, the last thing to do is to either consider already received acknowledgements for n_i's demand as invalid (since its demand is on hold) (done in Line 34) or to remember to ignore it when it comes (done in Line 36).
2) n'_i's demand has a lower priority but the competitors are authorized. In this case, n_i applies the pseudo-code in Lines 42-52. Note that in this case, n_i necessarily has an on-going request, otherwise it would have applied Lines 20-40. If n_i already sent its request to $n_{i'}$ (and thus $n_{i'}$is not in *reqs_to_send*), n_i withdraw temporarily its demand and, as before acts to ignore or remove the acknowledgement from $n_{i'}$. Then, an *ACK(renew)* is sent to n'_i.

If the situation falls in neither of the above cases, it means that n_i has a higher priority and the other group is not authorized. In this case, the acknowledgement is postponed by adding $n_{i'}$ in *waiting_reqs* (in Line 54).

5) Receiving an acknowledgement: Upon the reception of an akckowledgement, Node n_i applies the pseudo-code in Lines 58-76. n_i first tests whether one of the flags are set. If this is the case, it signals the end of the authorization of the other group: an end flag means that one node in the other group (n_j) left the critical section, and a $renew$ flag means that n_j is authorizing n_i's group to enter the critical section. In the particular case of end (see Lines 61-66), as

Algorithm 1 2-FGME algorithm (part 1).

```
1: procedure INITIALISATION
2:     req_id ←< n_i, ∞ >; state ← IDLE; LC_i ← 0
3:     waiting_reqs ← ∅
4:     reqs_to_send ← ∅
5:     acks_to_ign ← ∅
6:     competitors_authorized ← FALSE
7: end procedure

8: procedure REQUEST critical section
9:     state ← REQUESTING
10:    LC_i ++; req_id ← <n_i, LC_i>
11:    if ¬competitors_authorized then
12:        multicast Req(n_i, req_id) to competitors
13:    else
14:        reqs_to_send ← competitors
15:    end if
16:    acks_rcvd ← ∅
17: end procedure

18: procedure RECEIVE <Req(n_i', LC_i')>
19:    LC_i ← MAX(LC_i, LC_i')
20:    if req_id ≻< n_i', LC_i' > and state ≠ in_cs then
21:        send Ack(n_i) to n_i'
22:        if ¬competitors_authorized then
23:            competitors_authorized ← TRUE
24:            if req_id ≠< n_i, ∞ > then
25:                for all n_j|j≠i' ∈ competitors do
26:                    if n_j ∈ waiting_reqs then
27:                        waiting_reqs ← waiting_reqs \ {n_j}
28:                        send <Ack(n_i), renew> to n_j
29:                    else
30:                        send <End(n_i, LC_i)> to n_j
31:                    end if
32:                    reqs_to_send ← reqs_to_send ∪ {n_j}
33:                    if n_j ∈ acks_rcvd then
34:                        acks_rcvd ← acks_rcvd \ {n_j}
35:                    else
36:                        acks_to_ign ← acks_to_ign ∪ {n_j}
37:                    end if
38:                end for
39:            end if
40:        end if
41:    else
42:        if competitors_authorized and state ≠ in_cs then

43:            if n_i' ∉ reqs_to_send then
44:                send <End(n_i, LC_i)> to n_i'
45:                reqs_to_send ← reqs_to_send ∪ {n_i'}
46:                if n_i' ∈ acks_rcvd then
47:                    acks_rcvd ← acks_rcvd \ {n_i'}
48:                else
49:                    acks_to_ign ← acks_to_ign ∪ {n_i'}
50:                end if
51:            end if
52:            send <Ack(n_i), renew> to n_i'
53:        else
54:            waiting_reqs ← waiting_reqs ∪ {n_i'}
55:        end if
56:    end if
57: end procedure
```

the other group starts leaving the critical section, it is time for n_i to resume requesting the critical section: n_i multicasts its request to nodes in *reqs_to_send*. The Ack received is added to the list of acknowledgement received (except if it has to be ignored). Finally, if this Ack was the last to be received, n_i enters the critical section in Line 73.

6) Receiving a withdrawing/leaving notification: The *End* message is sent to notify that a node leaves the critical section or withdraw its demand to nodes that are not currently requesting it. Upon receiving such a message, a node applies Lines 77-86. The first thing to do is to no longer authorize the other group since one of its members withdraws its demand / leaves the critical section. Then, if a request from $n_{i'}$ was pending, it needs to be removed. Finally, as for the reception of a $Ack(end)$ message, n_i resumes its requesting phase.

7) Exiting the critical Section: Upon exiting the critical section, Node n_i applies Lines 87-98. It first reinitializes its state. Then, if requests were pending, it is time for n_i to send an acknowledgement to the nodes that issued them. For the other nodes, an *End* is sent.

IV. Experimental validation

In this section, we present a set of results obtained through real experimentation about the efficiency of the 2-FGME algorithm. We have evaluated the algorithm against three dimensions:

1) **concurrency**, the amount of nodes within a group able to enter the critical section at the same time;
2) **Network traffic induced**, the amount of messages generated by the protocol upon multiple simultaneous attempts at entering the critical section;
3) **latency**, the time taken by a node to manage to enter the critical section once it changed its state to *requesting*.

For the sake of comparison, we also evaluated the two algorithms proposed by Joung in [21], namely RA1, the simple extension Ricart-Agrawala algorithm to the case of Group Mutual Exclusion and RA2, enhanced with the notion of *captains* capturing nodes from the same group into critical section.

The software prototype used for the evaluations was developed in Java and relies on Apache Kafka [28] for message exchanges between nodes. Kafka is a high-level distributed publish-subscribe messaging middleware. It maintains feeds of messages in categories called topics. A process that publishes messages in a Kafka topic is called a *producer* and a process that subscribes to topics and processes messages is called a *consumer*. Kafka is a *broker* process, managing messages, and relieving the burden of messaging reliability from the programmer. Note that then, the following performance estimates partly depend on the performance delivered by Kafka. Note that the prototype developed is not a full-fledged Stream Processing Engine, as our focus was the

Algorithm 2 2-FGME algorithm (part 2).

58: **procedure** RECEIVE $<Ack(n_{i'}), renew, end>$
59: **if** end **or** $renew$ **then**
60: $competitors\_authorized \leftarrow$ FALSE
61: **if** end **then**
62: $acks\_to\_ign \leftarrow acks\_to\_ign \setminus \{n_{i'}\}$
63: **multicast** $<Req(n_i, req\_id)>$ **to** $reqs\_to\_send$
64: $reqs\_to\_send \leftarrow \emptyset$
65: $acks\_to\_ign \leftarrow acks\_to\_ign \setminus reqs\_to\_send$
66: **end if**
67: **end if**
68: **if** $n_{i'} \in acks\_to\_ign$ **then**
69: $acks\_to\_ign \leftarrow acks\_to\_ign \setminus \{n_{i'}\}$
70: **else**
71: $acks\_rcvd \leftarrow acks\_rcvd \cup \{n_{i'}\}$
72: **if** $acks\_rcvd = competitors$ **then**
73: $state \leftarrow$ IN_CS
74: **end if**
75: **end if**
76: **end procedure**

77: **procedure** RECEIVE $<End(n_{i'}, LC_{i'})>$
78: $LC_i \leftarrow max(LC_i, LC_{i'})$
79: $competitors\_authorized \leftarrow$ FALSE
80: **if** $n_{i'} \in waiting\_reqs$ **then**
81: $waiting\_reqs \leftarrow waiting\_reqs \setminus \{n_{i'}\}$
82: **end if**
83: **multicast** $<Req(n_i, req\_id)>$ **to** $reqs\_to\_send$
84: $reqs\_to\_send \leftarrow \emptyset$
85: $acks\_to\_ign \leftarrow acks\_to\_ign \setminus reqs\_to\_send$
86: **end procedure**

87: **procedure** EXIT critical section
88: $state \leftarrow$ IDLE
89: $req\_id \leftarrow < n_i, \infty >$
90: **if** $waiting\_reqs \neq \emptyset$ **then**
91: $competitors\_authorized \leftarrow$ TRUE
92: **multicast** $<Ack(n_i), end>$ **to** $waiting\_reqs$
93: **end if**
94: $others \leftarrow competitors \setminus waiting\_reqs$
95: **multicast** $<End(n_i, LC_i)>$ **to** $others$
96: $waiting\_reqs \leftarrow \emptyset$
97: $acks\_to\_ign \leftarrow \emptyset$
98: **end procedure**

validation of the scaling algorithm. In particular, the scaling procedure itself was not implemented. We refer the reader to our previous work for the experimentation of the associated decentralized scaling algorithm [6].

The experiments were conducted over nodes of Grid'5000 [3], the French national experimental computing platform. Each experiment was conducted on up to 4 nodes. A node include 2 Intel Xeon E5-2630 v3 with 8 cores each and 128 RAM interconnected by 2x10 Gbps network. For all of the following experiments, nodes are equally dispatched in the two groups.

A. Concurrency

In this section, we show the efficiency of 2-FGME in terms of concurrency and then compare it with the two

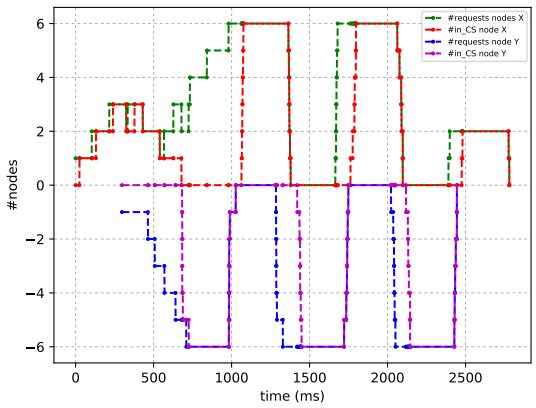

Figure 3. 2-FGME concurrency : number of nodes requesting the critical section compared to the number of nodes in the critical section for both groups (inCS = P = 300ms).

algorithms RA1 and RA2. By *concurrency*, we mean how many nodes of a same group succeed in entering the *critical section* when all nodes of all groups request it in a short time window.

For the experiments of this part, we set up a system of 12 nodes divided into 2 groups (6 nodes each). Let us denote P, the time between two requests emitted by a given node. In the philosophers' problem vocabulary, P is the time a philosopher, after leaving the *eating* state, stays in the *thinking* state, before entering the *requesting* state. Let us denote $inCS$, the time a node stays in the critical section (or in the *eating* state.) For the concurrency experiments, these two parameters were fixed to 300 ms.

Figure 3, Figure 4, and Figure 5, displays the concurrency level experimented for 2-FGME, RA2, and RA1, respectively. RA2 was placed before as it is our main competitor. In this section and, for the rest of the experiments, comparing with RA1 was performed for the sake of validation of our prototype and to confirm the concurrency superiority of both 2-FGME and RA2, as implemented within it.

Figures 3, 4, and 5 show the variation of the amount of nodes requesting and inside the critical section, respectively, for the two groups over time. For the sake of readability, each group was displayed on a different side of the Y-axis:

They show on the positive side of the Y-axis, the total number of nodes either requesting or in the critical section (green curve) and the number of nodes within them that are actually in the *critical section* (red curve) for the first group of nodes. Similarly, on the negative side of the Y-axis, figures show the same variables, but for the second group of nodes: The blue curve is for requesting nodes (or *in CS* nodes) and the purple curve shows nodes actually in *critical section*.

Starting with the 2-FGME and RA2 algorithms (Figures 3

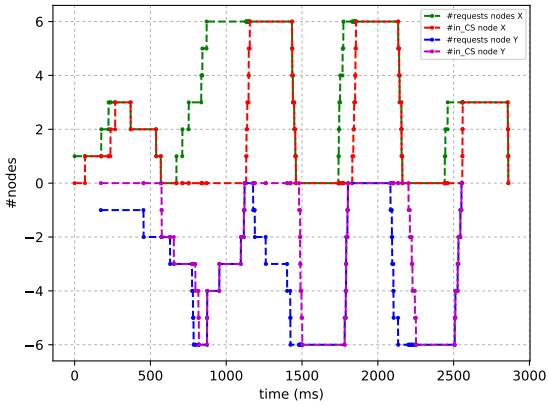

Figure 4. RA2 concurrency : number of nodes requesting the critical section compared to the number of nodes in the critical section for both groups (inCS = P = 300ms).

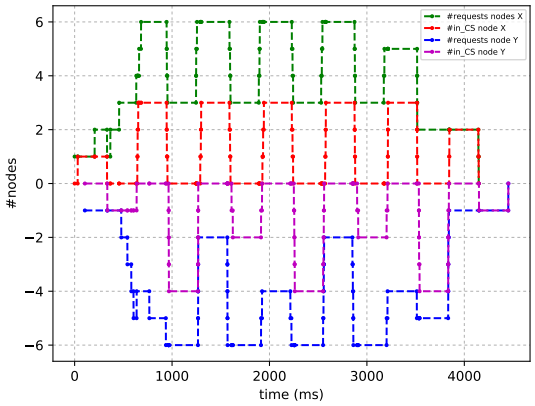

Figure 5. RA1 concurrency : number of nodes requesting the critical section compared to the number of nodes in the critical section for both groups (inCS = P = 300ms).

and 4), we observe that the concurrency offered by both algorithms is high as it reaches the highest performance possible where all nodes of the same group requested the *CS* reach it in a short window of time, and systematically, except for the first *round*. It shows that, in terms of concurrency, 2-FGME and RA2 offer a very similar level of performance.

On the other hand, Algorithm RA1's results, displayed in Figure 5, shows its limits in terms of concurrency, as during the experiment, only 3 nodes of the first group enter the *CS* over 6 and for the second group, 4 nodes over 6. This is to be expected, as already mentioned, as RA1 offers no mechanism to bypass the total order of the nodes established by Lamport's clocks.

It is to be noted that in the experiment whose results are given by Figure 3, nodes of the first group start requesting and entering the *CS* in the first part of the experiment, in the

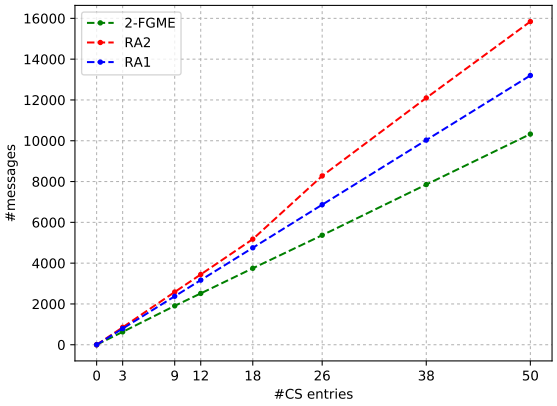

Figure 6. Traffic generated when increasing the number of entries in the critical section (12 nodes, P = inCS = 1s).

period of $time \in [0ms, 500ms]$ and remain requesting for the second part in the period of $time \in [500ms, 1000ms]$ but in that period nodes of the second group take over the entrance of the *CS* while nodes of the first group keep requesting the *CS*. From this point on, a sort of alternating scheme appears where each group fully enters critical section one after the other.

We can conclude that 2-FGME provides an optimized level of concurrency, very similar to the one provided by RA2. In particular, as RA2, it easily outperforms RA1. In the next section, we analyse the traffic performance, namely, the number of messages sent between nodes in the network.

B. *Traffic*

In this section, we focus on the number of messages sent between nodes in the network for the three algorithms. First, we start with an experiment where we fix the number of nodes to 12 and with the parameters $P = inCS = 1s$.

Figure 6 plots the number of messages generated by each protocol, when the number of times a node is set to enter the critical section increases. The red curve describes the behavior of RA2, the blue curve the behavior of RA1 and the green curve the behavior of 2-FGME. For each value of x, $f(x)$ is an average of 3 experiments. The first result of this figure is that all protocols generate a traffic which is clearly linear in the number of entries in the critical section. This is to be expected. The second, and most important takeaway of Figure 6 is that it shows that the traffic generated by 2-FGME is lower than the one generated by both RA1 and RA2.

Let us have a closer look: taking the example of the experiment with 18 entries, the average amount of generated messages of RA2 is 5176, on the other hand that of 2-FGME is 3748, which is a reduction by 27.58%. With 50 entries, the average number of generated messages by RA2 is 15843 but

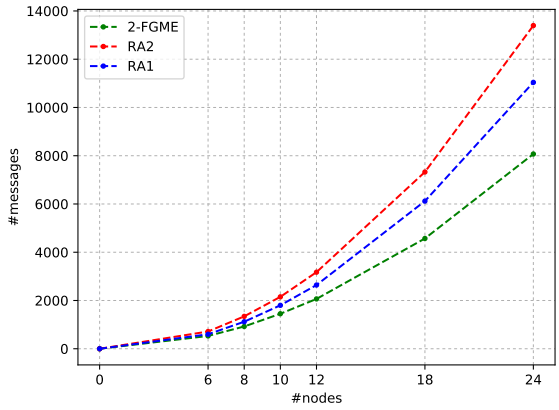

Figure 7. Traffic generated when increasing the number of nodes (10 entries in critical section, inCS = 1s).

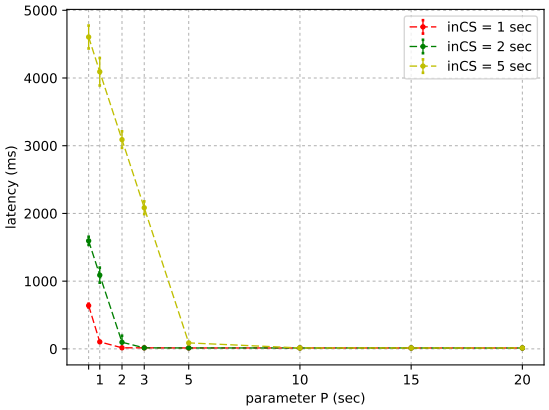

Figure 8. 2-FGME latency, 25 entries in critical section.

only 10327 messages sent by 2-FGME, which represents a gain of 34.81%. This is an important result, as it shows that, for 2 groups, 2-FGME allows a very similar concurrency level while significantly decreasing its cost.

For the following experiments, we fix now the entries frequency in the *critical section* to 10 and we vary the number of nodes from 6 to 24. And we keep the same parameters like before $P = inCS = 1s$.

Figure 7 plots the number of messages of 2-FGME (green curve), RA1 (blue curve) and RA2 (red curve) generated when increasing the total number of nodes in the groups (remind that nodes are equally dispatched amongst groups).

Similarly to what has been observed before in Figure 6, we see that 2-FGME generates less messages than both RA1 and RA2. As an example, the average of messages sent with 12 nodes (6 nodes per group) of 2-FGME is 2065 while RA2 and RA1 generate 3445 and 2640 messages respectively. As far as with 24 nodes (12 nodes per group), the 2-FGME generates 8071 messages while RA2 generate 13395. In other words, a 40% reduction in the traffic is brought about by 2-FGME (for similar concurrency levels) for the case of two groups.

Notice that the curves in Figure 7 are the expression of a quadratic behavior. Let us take the simple example of RA1. We can easily model its traffic generation behavior. Let us denote n as the total number of nodes and y the number of entries in critical section. Let us assume that the number of entries is the same for all nodes in the end of the experiment and the nodes are dispatched equally into groups. In this case, the number of messages follow the function $f(n, y) = 2 * y * (n^2 - 1)$. Even if the complexity of the algorithms of 2-FGME and RA2 differs from RA1, their curves follow a similar global behavior, and 2-FGME offers a significant reduction factor.

As a conclusion, Algorithm 2-FGME is more efficient in terms of network traffic and this aspect is important especially in the context of *stream processing* where any delay can affect the global performance of the system.

In the next section, we conclude this experimental study with an analysis of the latency experimented by nodes when they start requesting the critical section.

C. Latency

In this section, we analyse the average delay for a node to enter the critical section since it first sent its request. For the following experiments, we used 12 nodes and each of them enters in the critical section 25 times. To calculate each point in the curve, we only kept the latencies for the last 20 entries (to get rid of any initialization or stabilization effect, especially regarding the underlying Kafka broker), and averaged them. Each experiment being repeated 3 times, each point is an average obtained over 60 values.

Figure 8, Figure 9 and Figure10 plot the latency as a function of the parameter P, the time spent inside the critical section $inCS$ being fixed. The graph is composed of three curves: the yellow curve is for $inCS = 5sec$, the green curve is for $inCS = 2sec$, and the red curve is for $inCS = 1sec$.

Let us first focus on the results of Figure 8 in which we analyse the latency of 2-FGME. For each curve, we notice two parts: the first part, where $P < inCS$ and the second one, where $P > inCS$.

During the first part, in which $P < inCS$, we can see that the curve $f(P)$ is falling linearly from $P = 0$ to $P = inCS$. This reflects the fact that in this configuration, the latency experienced by a node is mostly composed of the time it takes for the nodes in the other group to leave the critical section.

In the second part of the curves where $P \geq inCS$, we observe a very low and stable latency (in average $f(P) \approx 13ms$). This shows that globally, the extra time

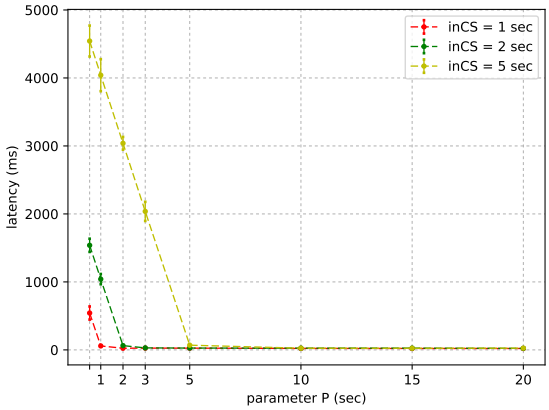

Figure 9. RA2 latency, 25 entries in critical section.

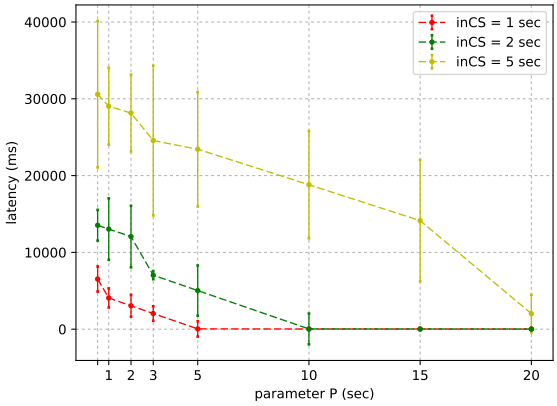

Figure 10. RA1 latency, 25 entries in critical section.

brought about by the contention of the algorithm is negligible compared to the time taken by nodes to leave the critical section (delay which can not be reduced.)

Figure 9 shows the same observation for RA2, with similar values of $f(P)$. This is to be expected in the sense that, even if permissions to enter the critical section are granted by different nodes (the nodes within the same group for RA2, the nodes of the competitor group for 2-FGME), they lead to globally the same kind of behaviours.

Finally, Figure 10 shows the same behavior for RA1, but with different values compared to 2-FGME and RA2. The values are higher in this case. This can be explained by the fact that the concurrency of RA1 being lower than 2-FGME and RA2 (as presented before in Figure 5), therefore, nodes of a group requesting an entry to the *critical section* may not enter in critical section as quickly as for RA2 and 2-FGME. This is illustrating by the example displayed in Figure 5 where all nodes of the two groups request the *critical section*: Nodes of the first group may not enter the CS while other nodes of the same group succeed, in which case, the former nodes will have to wait for nodes in their own group to leave the critical section, and then wait again that nodes of the other groups leave on their turn before being able to enter critical section themselves. This also explains the higher variability of latency to enter the critical section, as illustrated by the standard variation included in this last series of curve.

V. Conclusion

This paper described a new algorithm to solve the problem of Group Mutual Exclusion, with specific constraints inherited from the context of stream processing. This algorithm, called 2-FGME, focus on the particular case of having two concurrent groups of nodes wishing to enter in their critical section. In the particular context of decentralized stream processing, mutual exclusion is required to avoid two neighbouring groups of operators scaling at the same time, to avoid hazardous decentralized graph updates.

Because focused on two groups only, our algorithm is able to exhibit a reduced message complexity compared to the similar algorithms found in literature while offering a very similar level of concurrent occupancy of the critical section. These results were obtained through real experimentation of our 2-FGME algorithm and its comparison with two of the most classically used algorithm for group mutual exclusion.

Future work will mostly consist in providing a formal proof of our approach, and extending it to the general case of an unlimited number of groups. Also, we plan to compare our approach with other possible schemes for mutual exclusion based on hierarchical and quorum-based algorithms, in terms of the overhead they generate.

Acknowledgment

This project was partially funded by ANR grant ASTRID SESAME ANR-16-ASTR-0026-02.

References

[1] D Agarwal and AE Abbadi. An Efficient Solution to the Distributed Mutual Exclusion Problem. In *Proceedings of the 8th Principles of Distributed Computing Conference (PODC)*, 1989.

[2] R. Atreya, N. Mittal, and S. Peri. A Quorum-Based Group Mutual Exclusion Algorithm for a Distributed System with Dynamic Group Set. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1345–1360, 2007.

[3] Daniel et al. Balouek. Adding virtualization capabilities to the Grid'5000 testbed. In *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer, 2013.

[4] Joffroy Beauquier, Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Group mutual exclusion in tree networks. In *9th International Conference on Parallel and Distributed Systems, ICPADS 2002, Taiwan, ROC, December 17-20, 2002*, pages 111–116. IEEE Computer Society, 2002.

[5] Mehdi Mokhtar Belkhiria and Cédric Tedeschi. A Fully Decentralized Autoscaling Algorithm for Stream Processing Applications. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26-30, 2019, Revised Selected Papers*, volume 11997 of *Lecture Notes in Computer Science*, pages 42–53. Springer, 2019.

[6] Mehdi Mokhtar Belkhiria and Cédric Tedeschi. Design and evaluation of decentralized scaling mechanisms for stream processing. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Sydney, Australia, December 11-13, 2019*, pages 247–254. IEEE, 2019.

[7] Nicolò M. Calcavecchia, Bogdan A. Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. DEPAS: a Decentralized Probabilistic Algorithm for Auto-scaling. *Computing*, 94(8):701–730.

[8] Sébastien Cantarell, Ajoy Kumar Datta, and Franck Petit. Self-stabilizing atomicity refinement allowing neighborhood concurrency. In Shing-Tsaan Huang and Ted Herman, editors, *Self-Stabilizing Systems, 6th International Symposium, SSS 2003, San Francisco, CA, USA, June 24-25, 2003, Proceedings*, volume 2704 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2003.

[9] Sébastien Cantarell, Ajoy Kumar Datta, Franck Petit, and Vincent Villain. Group mutual exclusion in token rings. *Comput. J.*, 48(2):239–252, 2005.

[10] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed QoS-aware Scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 344–347, New York, NY, USA, 2015. ACM.

[11] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Decentralized self-adaptation for elastic data stream processing. *Future Generation Comp. Syst.*, 87:171–185, 2018.

[12] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM SIGMOD'13*, pages 725–736, 2013.

[13] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[14] Pierre-Jacques Courtois, Frans Heymans, and David Lorge Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[15] Edsger W Dijkstra. hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. Springer, 1971.

[16] Edsger W Dijkstra. Two Starvation Free Solutions to a General Exclusion Problem. *Unpublished Tech. Note EWD*, 625, 1978.

[17] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.*, 25(6), 2014.

[18] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec 2012.

[19] Jean-Michel Hélary, Noël Plouzeau, and Michel Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *Comput. J.*, 31(4):289–295, 1988.

[20] Yuh-Jzer Joung. Asynchronous group mutual exclusion. *Distributed computing*, 13(4):189–206, 2000.

[21] Yuh-Jzer Joung. The Congenial Talking Philosophers Problem in Computer Networks. *Distributed Computing*, 15(3):155–175, 2002.

[22] Patrick Keane and Mark Moir. Simple Local-Spin Group Mutual Exclusion Algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 12:673 – 685, 08 2001.

[23] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[24] Gérard Le Lann. Distributed Systems-Towards a Formal Approach. In *IFIP congress*, volume 7, pages 155–160. Toronto, 1977.

[25] Mamoru Maekawa. A $\sqrt{(N)}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2):145–159, 1985.

[26] Alain J Martin. Distributed mutual exclusion on a ring of processes. *Science of Computer Programming*, 5:265–276, 1985.

[27] Glenn Ricart and Ashok K Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, 24(1):9–17, 1981.

[28] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. Building a replicated logging system with apache kafka. *PVLDB*, 8(12):1654–1655, 2015.

[29] Li Wang, Tom Z. J. Fu, Richard T. B. Ma, Marianne Winslett, and Zhenjie Zhang. Elasticutor: Rapid elasticity for realtime stateful stream processing. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 573–588. ACM, 2019.

[30] K-P Wu and Y-J Joung. Asynchronous Group Mutual Exclusion in Ring Networks. *IEE Proceedings-Computers and Digital Techniques*, 147(1):1–8, 2000.

[31] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: Traffic-aware online scheduling in storm. In *IEEE 34th International Conference on Distributed Computing Systems*, 2014.

[32] Yuh-Jzer Joung. Quorum-Based Algorithms for Group Mutual Exclusion. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):463–476, 2003.